NPS52-81-006

NAVAL POSTGRADUATE SCHOOL

Monterey, California

VALUES AND OBJECTS IN PROGRAMMING LANGUAGES

Bruce J. MacLennan

April 1981

Prepared for:

Chief of Naval Research
Arlington, Va. 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Acting Provost

The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-81-006	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Values and Objects in Programming Languages		5. TYPE OF REPORT & PERIOD COVERED Technical 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE April 1981 13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Values, Objects, Value-oriented, Object-oriented, Applicative, Imperative, Programming style, Pointer, Variable-free, Instances, Names

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The terms value-oriented and object-oriented are used to describe both pro-programming languages and programming styles. This paper will describe the differences between values and objects and to show that their proper discrimination can be a valuable aid to conquering program complexity. The first section will show that values amount to timeless abstractions for which the concepts of updating, sharing and instantiation have no meaning. The second section will show that objects exist in time and, hence, can be created, destroyed, copied, shared and updated. The third section shows that proper discrimination of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20.

these concepts in programming languages will clarify problems such as the ro
of state in functional programming. The paper concludes by demonstrating th
use of the value/object distinction as a tool for program organization.

VALUES AND OBJECTS IN PROGRAMMING LANGUAGES*

B. J. MacLennan

81/04/13

1. INTRODUCTION

The terms value-oriented and object-oriented are used to describe both programming languages and programming styles. This paper will describe the differences between values and objects and to show that their proper discrimination can be a valuable aid to conquering program complexity. The first section will show that values amount to timeless abstractions for which the concepts of updating, sharing and instantiation have no meaning. The second section will show that objects exist in time and, hence, can be created, destroyed, copied, shared and updated. The third section shows that proper discrimination of these concepts in programming languages will clarify problems such as the role of state in functional programming. The paper concludes by demonstrating the use of the value/object distinction as a tool for program organization.

* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

2. VALUES

Values are applicative. The term value-oriented is most often used in conjunction with applicative programming, that is, with programming with pure expressions and without the use of assignment or other imperative facilities. Another way to put this is that value-oriented programming is programming in the absence of side-effects. This style of programming is important because it has many of the advantages of simple algebraic expressions, viz. that an expression can be understood by understanding its constituents, that the meaning of the subexpressions is independent of their context, and that there are simple interfaces between the parts of the expression that are obvious from the syntax of the expression. For instance, in 1 + (x-y)/z, the meaning of the entire expression is only dependent on the meanings of 1 and (x-y)/z, the meaning of (x-y) is independent of 1 + .../z, and the relation between (x-y) and the rest of the expression is obvious from the form of the expression.

Primitive values are the paradigms. The idea of values is most familiar, as the above discussion indicates, in the realm of numbers. Since numbers have no internal structure, they can only be combined by the operations (such as +) to yield other numbers. In programming languages the primitive, or atomic, data such as integers, reals, Booleans and characters, are usually treated as values. These are all operated on by side-effect-free operations

that yield other values. Expressions involving values satisfy all of the desirable properties listed above (provided they contain no side-effect producing procedures). These primitive data values provide a model for all other values.

Compound values are also possible. It might seem that values are necessarily restricted to primitive data values, since a compound data value immediately admits the possibility of updating a component of the data value. For instance, arrays cannot be considered values because of the possibility of updating one component of the array. This is not essential to the concept of arrays, however, as we can see by looking at APL. In this language it is possible to perform extensive computations with arrays without ever performing an assignment. This is because the array is treated as a whole and, instead of updating a component of an array, an entire new array is computed instead. To put it another way, in APL expressions arrays are treated as values. Of course, this is not true in APL if one uses the assignment operation, which does allow updating of array components. A subset of APL without the assignment operator is a value-oriented language.

Another example of a compound value is a complex number. As used by mathematicians, these values are operated on by side-effect-free operations that produce other complex numbers. Of course, it is possible to extract the real and imaginary parts of a complex number, or to put two reals together to make a complex

number. What is not possible is the selective updating of one part of a complex number analogous to the selective updating of an array component.

Pure LISP is another example of a value-oriented language. That is, if the PROG feature, EQ, and the RPLACA and RPLACD procedures are not used, then LISP lists can be treated as mathematical objects and LISP functions have all the desirable properties of arithmetic operations. (PROG and the RPLACx functions allow updating; EQ allows instances to be distinguished)

Hoare, in his Recursive Data Structures, has described how these ideas can be extended to more general structures. In this paper Hoare advocates eliminating the pointer from programming languages in favor of recursive definitions of data types. We will see later that recursive data types are a natural concommitant of values and that pointers are closely connected with objects. Hoare shows that if data structures are properly restricted then all pointer manipulation can be performed automatically. This is advantageous because pointers are a common source of errors, a point elaborated below.

Values are more secure. Consider an expression such as (-x)/(x+y). There is no chance that the operation (-x) on x will effect the outcome of (x+y). This is because the thing denoted by x is a value and the '-' operation has no side-effects. That is, each part of an expression involving values is independent of

all the others. One reason for this is that values are read-only, i.e., it is not possible to update their components. Since they are unchangable, it is always safe to share values for efficiency; there is never any danger of one expression altering something which is used by another expression. Any sharing that takes place is hidden from the programmer and is done by the system for more efficienct storage utilization. For example, LISP systems allow final portions of lists to be shared among any number of lists. This can never lead to surprising results when dealing with values because updating is not allowed. Avoiding updating eliminates dangling reference problems and simplifies deallocation.

What are values? We have discussed a number of properties of values and have presented several examples of them. What exactly are they? The best examples of values are mathematical entities, such as integers and real and complex numbers, hence we may expect to understand values better by understanding these better.

One characteristic of mathematical entities is that they are eternal, in the literal sense of being out of time. To put it another way, the concept of time or duration does not apply to mathematical entities any more than the concept color applies to them; they are neither created nor destroyed. When we write 2+3 there is no implication that 5 has just come into existence and that 2 and 3 have been consumed; mathematical entities are

timeless. Similarly, numbers are immutable. When we perform the operation (1+2i)+5 and get 6+2i, we have not changed 1+2i in any way. It is not as though we had changed its real part from 1 to 6. What is it about numbers that give them these properties?

Values are abstractions. The fundamental fact that gives mathematical entities and other values these properties is that they are abstractions, or universals, or concepts. Although a full explication of mathematical entities is beyond the scope of this paper it should be fairly clear that the number 2 is an abstraction that subsumes all instances of "two-ness." That is, just as this tree or that tree or another tree are all subsumed under the concept 'tree', so also this pair or that pair or this other pair are all subsumed under the concept 2. Abstractions are universal; the concept 2 subsumes all possible pairs, those existing, those in our imagination, those that have existed, those that are yet to exist. This universal nature of abstractions makes them eternal, or out of time. The number 2 can neither be created nor destroyed because its existence is not tied to the creation or destruction of particular pairs. Indeed, the concept of existence, in its usual sense, is not applicable to the number 2. It is the same with all values, because all values correspond to abstractions; they can neither be created nor destroyed.

It is also the case that abstractions, and hence values, are immutable. Suppose there is $2 in someone's bank account, we may

say there is a pair of dollars in his account. This pair of dollars is subsumed by the abstraction (value) 2. Now suppose that we increase the balance to $3. Have we changed the value 2? No, there is now a triple of dollars in his account that is subsumed by the value 3, but the value 2 is still unchanged; it still subsumes all the pairs, including that particular pair of dollars that no longer exists. Thus, although values may be operated on, in the sense of relating values to other values, they cannot be altered. That is, 2+1=3 states a relation among values; it does not alter them. Values are eternal and immutable.

There is a possible confusion that must be prevented. When in a programming language we assign x the value 2, x:=2, and later add one to x, x:=x+1, haven't we changed a number, which is a value? No, we haven't; the number 2 has remained the same. What we have changed is the number that the name x denotes. We can give names to values and we can change the names that we give to values, but this doesn't change the values. The naming of values and the changing of names is discussed in a later section.

Values cannot be counted. A corollary of the above is that there is not such thing as "copies" of a value. This should be clear from mathematics: it is not meaningful to speak of this 2 or that 2, there is just 2. We say that the number 2 is uniquely determined by its value. This is because an abstraction is uniquely determined by the things which it subsumes, hence, anything which subsumes all possible pairs is the abstraction 2.

Therefore, the concept 'number' is not even applicable to abstractions; it makes no sense to ask how many 2's there are. While it may be meaningful in a programming language to make another copy of an array, it is pointless to make another copy of a value; there is no such thing. There is also no reason to make such a copy since values are immutable. (It is, of course, possible to make copies of a representation of a value; this is discussed later.)

It is also meaningless to talk about the sharing of values. Since values are immutable, and can neither be counted nor copied, it is irrelevent whether different program segments share the same value or different "copies" of the value. Of course, there may be implementation differences. If a long string value is assigned to a variable it will make a big difference whether a fresh copy must be made or whether a pointer to the original copy can be stored. While this is an important implementation concern, it is irrelevent to the semantics of values.

Values are used to model abstractions. We have discussed a number of the characteristics of values but have not discussed whether values should be included in programming languages, or, if they are, what they should be used for. The answer to this question lies in the relation we have shown between values and abstractions: values are the programming language equivalent of abstractions. Thus, values will be most effective when they are used to model abstractions in the problem to be solved. This is

in fact their usual use, since integer and real data values are used to model quantities represented by integer and real numbers. Similarly, the abstraction 'color' may be modeled by values of a Pascal or Ada enumeration type, (RED, BLUE, GREEN). On the other hand, it is not common to treat compound data values, such as complex numbers or sequences, as values. If done, this would eliminate one source of errors. Now, value-oriented languages, such as the languages for data-flow machines and functional programming, have only values. Is there any need for objects at all? This is answered in the following sections.

3. OBJECTS

Computing can be viewed as simulation. It has been said that computing can be viewed as simulation. This is certainly obvious in the case of programs that explicitly simulate or model some physical situation. The metaphor can be extended to many other situations. Consider an employee data base; each record in the data base corresponds to an employee. The data base can be said to be a simulation, or model, of some aspects of the corporation. Similarly, the data structures in an operating system often reflect the status of some objects in the real world. For instance, they may reflect the fact that a tape drive is rewinding or in a parity-error status. The data structures can also reflect logical situations, such as the fact that a tape drive is assigned to a particular job in the system.

A data structure is needed for each entity. It should be clear that simulation is simplified if there is a data structure corresponding to each entity to be simulated; this simplifies the programming of the simulation by factoring and encapsulating related information. This is exactly the approach that has been taken in object-oriented programming languages, such as Smalltalk. The usual way to structure a program in such a language is to create an object for each entity in the system being modeled. These may be real-world objects or objects that are only real to the application, such as figures on a display screen. The messages these objects respond to are just the relevent manipulations that can be performed on the corresponding real entities. Given this relationship between programming language objects and real world objects, we will try to clarify the notion of an object.

What is an object? In our programming environment we have objects and in the real world we have objects. Just what is an objct? When we attempt to answer this question we immediately find ourselves immersed in age-old philosophical problems. In particular, what makes one object different from another? We can imagine another typewriter that looks exactly like the one before us now, sharing every characteristic and attribute of this typewriter, except being different. That is, it is easy to imagine two instances of the same typewriter. What makes them different typewriters? One philosophical answer to this question is to say that while the two typewriters have the same form, they have

different substance. To put this is more concrete terms, we could say that the two typewriters are alike in every way except that they occupy different regions of space. Now, we find exactly the same situation arising in programming languages. We may have two arrays that contain exactly the same values, yet they are two different arrays. What make them different? We would say that they occupy different locations. So by analogy, the form of the array is the order and value of its elements while the substance of the array is the region of memory it occupies.

This is a philosophical problem. There is also a less philosophical way in which we distinguish real world objects: we give them proper names. For instance we can use Sam and Joe as the names of identical twins. Even though each of these entities is (we can presume) alike in all of its other attributes, they can always be distinguished by their names. We find an exactly analogous situation in programming languages. Programming objects, such as the arrays already mentioned, generally have a unique name: the reference to the object. Tdis is generally closely related to the region of storage the object occupies. This is not necessary, however, as we can see by consdering a file system. It is easy to see that files are objects; it is quite normal to have two different files with the same contents. Of course, if the files are to be distinguished, then they must have distinct names. Incidently, it will also be the case that they occupy distinct areas of the storage system a fact that we

can test by updating one file and observing that the other doesn't change. Now, for our purposes, we will not be too concerned about what individuating element is used; whether it is some form of unique identification (such as a capability), or whether it is implicit in the region of storage occupied; we will assume that each object is different from every other object even if they contain the same data values. Thus there may be any number of instances of otherwise identical objects. This leads to a number of further consequences.

Objects can be changed. Now we have said that the identity of an object is independent of any of its internal properties or attributes. For instance, even if all of the elements of an array are changed, it is still the same array (because it occupies the same region of storage). This is of course like real world objects, for they too can change and retain their identity. Values, on the other hand can never change. For instance, if we add 5 to 1+2i, we don't change 1+2i, we produce a new value, 6+2i. This changability, this fact that an object may have one set of properties at one time and a different set at another time, is a distinctive feature of objects (and of programs).

Objects have state. This changability of objects leads to the idea of the state of an object, the sum total of the internal properties and attributes of an object at a given point in time. Thus, we can say that the state of an object may be changed in time. State is of course a central idea in computer science, so

it is not surprising to find that objects are at the heart of computer science. Since the state of an object can change in time, it is certainly the case that objects exist "in time", i.e., they are not eternal like values.

Objects can be created and destroyed. The fact that objects are not eternal leads to the conclusion that they can be both created and destroyed. This is familiar in programming languages where, for instance, an array may be created every time a certain block is entered and destroyed every time it is exited. Many languages also provide explicit means for creating and destroying objects (e.g., Pascal's 'new' and 'dispose'). Since values are eternal, it is meaningless to speak of them being either created or destroyed.

Objects can be abandoned. In many computer systems and programming languages it is possible for an object to be abandoned. That is it may be possible for all possible paths so that object, and all external relations with that object to be destroyed. When this occurs it is irrelevent whether the object exists or not and it can be destroyed automatically, i.e., it can be reclaimed by garbage collection. This is only possible if it is not possible to "search" for the object by computing and trying out references, or by looking for it the way we might look for a real object, by searching all possible locations. It is also necessary to consider all external relations in deciding whether the object has been abandoned. An object that has a correlate on

a display screen has an external relation, namely its relation to that shape on the screen. It is not possible to garbage collect it even though there may be no references to the object within the computer system. The object may continue to act independently on the display screen. (This would not usually arise in a real computer system since a potentially active object would almost always be related to some other object, e.g., a scheduler table).

Objects can be shared. Since there can be any number of instances of otherwise identical objects and since objects can change their properties in time, it is a crucial question whether an object is shared or not. Suppose A and B are two names for the same object: an array with the element values (5, 3, 8). If one subprogram changes the second element of A to 4, then both A and B will now name objects with the values (5, 4, 8). This is because A and B are two names for the same object and the object is still shared even if its properties have changed. Hence, if subprogram P called the object A and subprogram Q called the object B, then any changes made to the object A by P would be seen by Q as changes to B. The object is shared. Conversely, if A and B were the names of two different objects that at some point in time both happened to have the values (5, 3, 8), then a change to A would have no effect on the object B; they are two distinct objects. Such side-effects are common in programming and are often used by programmers as a way of communicating. People also frequently use shared objects as means of

communication. For instance, two persons may communicate by altering the state of a blackboard.

Recall that in our discussion of values we found that the issue of sharing didn't apply. Whether a particular implementation chooses to share copies of values or not is irrelevent to the semantics of the program; it is strictly an issue of efficiency. Sharing is a crucial issue where objects are concerned.

Computer science as objectified mathematics. We can see now an important difference between the domain of mathematics and the domain of computer science. Mathematics deals with things such as numbers, functions, vectors, groups, etc. These are all abstractions, i.e., values. It has been said that the theorems of mathematics are timeless, and this is literally true. Since mathematics deals with the relations among values and since values are eternal, the resulting relations (which are themselves abstractions and values) are eternal. Conversely, much of computer science deals with objects and with the way they change in time. State is a central idea. It may not be unreasonable to call computer science objectified mathematics, or object-oriented mathematics.

It has frequently been observed that the advantage of applicative programming is that it is more mathematical and eliminates the idea of state from programming. We can see that this means that applicative programming deals only with values (indeed,

several languages for applicative programming are called "value-oriented" languages). Really, applicative programming is just mathematics.

These ideas can be summarized in two observations:

:: Programming is object-oriented mathematics.

:: Mathematics is value-oriented programming.

These two principles show the unity between the two fields and isolate their differences.

VALUES AND OBJECTS IN PROGRAMMING LANGUAGES

Most languages confuse them. It should be clear from the examples that we have used in the previous sections that both values and objects are accomodated in most programming languages. This accomodation is usually very asymmetric and ad hoc, however. For instance, a language such as FORTRAN supports values of several types, including integers, reals, complex numbers and logical values. These are all treated as mathematical values; for instance, it is not possible to "update" the real part of a complex number separately from the imaginary part. Of course, it is possible to store all of these values in variables, but that is a different issue, as we will see later. On the other hand, FORTRAN provides objects in the form of updatable, sharable arrays. This pattern has been followed with few variations in most other languages. All of these languages unnecessarily tie

the value or object nature of a thing to its type, usually by treating the atomic data types as values and the compound data types as objects. We will argue below, that this confusion complicates programming.

Programming languages are most often deficient in their treatment of compound values; in particular, they rarely provide recursive data types as Hoare described them (Recursive Data Structures). They tend to confuse the logical issue of whether a thing should be an object (i.e., it is shared, updatable, destroyable, etc.) with the implementation issue of whether it should be shared for efficiency. We will see how this can be solved later.

Mathematics deals poorly with objects. We have said that mathematics is value-oriented; that is it deals with timeless relations and operations on abstractions. Concepts that are central to objects (and computer science), such as state, updating and sharing, are alien to mathematics. This is not to say that it is impossible to deal with objects in mathematics; it is done every day, only awkwardly. For instance, it is common to deal in physics with systems that change in time; they are represented mathematically by functions of an independent variable representing time. The relationships between objects can be represented as differential equations (or difference equations if state changes are quantized). Similarly, mathematics can distinguish instances of an object by attaching a unique name (generally a

natural number) to each instance of a value. For instance, complex objects can be represented by pairs (n, z), where n is a number, the object's "name", and z is a complex number. Then (n, z) and (m, z) are two different instances of the complex number z if n=m. Also, (n, z) and (n, z') represent two different states of the same object n. These techniques work but are awkward. A more fully developed attempt to apply the concepts of mathematics to the description of objects can be seen in denotational sematics. Here the state is explicitly passed from function to function to represent its alteration in time.

Fen theory deals poorly with values. In our fen theory (MacLennan, 1973) we attempted to deal with these problems by developing an axiomatic theory of objects. This was done in two ways: (1) the axiom of set theory that forces two sets with the same values to be identical was discarded. This permitted multiple instances of the same set. (2) An axiom was inserted that required there to be at least a countable infinity of instances of each set. The result was an object-oriented theory of sets and relations. This worked well for describing many of the properties of objects and for defining the semantics of those programming language constructs that are object-oriented. Unfortunately, it suffered from the dual problem of mathematics: it was awkward to deal with values. For instance, the complex number 1+2i would be represented by instances of the relation (re:1, im:2). But this immediately led to problems. How many instances should we let correspond to 1+2i, just one prticular

one, or all of the ones with that structure? And what is 2? Is it the name of some distinguished object that we have chosen to represent 2 or does it denote any object with a certain structure? There are related problems with operations on values. For instance, which 5 does 2+3 return? These are all problems of attempting to deal with values in an object-oriented system. Values are inherently extensional while fen theory is inherently intensional (see A Dictionary of Philosophy, p. 109). The solution adopted in fen theory was to treat values as equivalence classes of objects in the supporting logic. This was possible because that logic was extensional (i.e., value-oriented).

Computers use objects to represent values. These are exactly the problems that must be faced in dealing with values on a computer. Abstractions are not physical objects (except so far as they exist in our brains), so to deal with them they must be represented or encoded into objects. We do this when we represent the number 2 by the numeral '2' or the word 'two' on a piece of paper. Once a value has been represented as an object it acquires some of the attributes of objects. For instance, this '2' is a different instance of the numeral '2' from that in the previous sentence, and from the second one in this sentence. Clearly, whenever a value is to be manipulated in a computer it must be represented as the state of some physical object. Typically, there will be many such representing objects in a computer at a time. For instance, 2 may be represented by a bit pattern in a register and in several memory locations. Therefore,

everything "in" a computer is an object; there are no values in computers. This does not imply, however, that values should be discarded from programming languages.

Programmers need values. From the programmer's viewpoint there are both values and objects. Of course, a purely object-oriented programming language could be designed. This could be done by storing everything in memory and then only dealing with the addresses of these things. It would be like having a pointer to every object. It would then be necessary for the programmer to keep track of the different instances of what were intuhtively the same value so that he wouldn't accidently update a shared value or miss considering as equal two instances of the same value. Some languages actually come close to this, such as LISP (with EQ and the RPLACx procedures) and Smalltalk. Unless such a language were carefully designed, it would be almost impossible to deal with values such as numbers in the usual way.

Programmers need objects. Conversely, programmers need objects in their programming languages. There have, for sure, been completely value-oriented programming languages. These include the FP and FFP systems of Backus (Backus, 1978). It is interesting to note, however, that Backus went on to define the AST system, which included the notion of state (and implicitly, of objects). Applicative languages were originally developed in reaction to what was surely an overuse of objects and imperative features in programming. Yet, it seems clear that we cannot

eliminate them from programming without great inconvenience. It is not uncommon to see applicative programs pass large data structures that represent the state of the computation from one function to the next. The result in such a case is not greater clarity, but less. We should not be surprised to have to deal with objects in programming; as we agued before, this is a natural outgrowth of the fact that we are frequently modeling real world objects. A better solution than banning objects is to determine their proper application and discipline their use.

We should use appropriate modeling tools. Programmers should be clear about what they are trying to model and then use the appropriate constructs. If they are modeling an abstraction, such as a number, then they should use values; if they are modeling an entity or thing that exists in time, then they should use an object. This implies that languages should support both values and objects and the means to use them in these ways. To put it another way, we must develop an appropriate discipline for using values and objects and linguistic means for supporting that discipline.

Names should be fixed. How can we arrive at such a discipline? How can we tame the state? One of the motivations for value-oriented programming is the incredible complexity that can result from a state composed of hundreds or thousands of individual variables, all capable of being changed (the Von Newmann bottleneck). We can see a possible solution to this problem by

looking at natural languages. Generally, a word has a fixed meaning within a given context. This holds whether the word is a common noun or a proper name. For instance, the word 'tree' invariably (with multiple meanings differentiated by context) refers to a particular abstraction. Similarly, the name 'Aristotle' refers to a particular object (one no longer in existence, in this case). We do not use 'tree' to refer to one abstraction one moment and another the next, or 'Aristotle' to refer to one object one moment and another the next. Yet this is exactly what we do with variables in programming languages. To the extent that we need temporary identifiers, natural languages provide pronouns. These are automatically bound and have a very limited scope (generally a sentence or two).

Can these ideas be applied to programming languages? It would seem so; let's consider the consequences. Suppose that names in programming languages were always bound to a fixed value or object within a context; effectively all names would be constants. For instance, we could give a name, such as 'pi', to a value, and we would be able to use built-in names for values, such as '2'. Similarly, whenever an object was created it could be given a name (e.g., 'Gibralter') that would refer to that object until it was destroyed. There would be no "variables" that can be rebound from moment to moment by an assignment statement. Variables in the usual sense would only be allowed as components of the state of an object and the only allowable assignments would be to these components.

Would it be possible to program in such a language, or would it be too inconvenient? Without actually designing it, it is difficult to tell. We can only point to the fact that a considerable amount of good mathematics has been done without the aid of variables, not to mention a considerable volume dealing with real world objects. Such a language could provide, as does mathematics, mechanisms for declaring constants of very local scope. Some languages do provide these mechanisms already (e.g., let t = (a+b)/2 in ..., or sin(t)+cos(t) where t = ...). As suggested by natural languages, it might be possible to provide some sort of pronoun facility. Hence, what we are describing is a programming language that is variable-free, but does not do away with objects, values, or names.

CONCLUSIONS

In this paper we have distinguished the two concepts 'value' and 'object'. We have shown that values are abstractions, and hence eternal, unchangable and non-instantiated. We have shown that objects correspond to real world entities, and hence exist in time, are changable, have state, are instantiated, and can be created, destroyed, and shared. These concepts are implicit in most programming languages, but are not well delimited.

We claim that programs can be made more managable by recognizing explicitly the value/object distinction. This can be done by incorporating facilities for handling values and objects in

programming languages.

REFERENCES

Backus, J., Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, CACM 21, 8, August 1978, pp. 613-641.

Flew, A., A Dictionary of Philosophy, St. Martin's Press, New York, 1979.

Hoare, C.A.R., Recursive Data Structures, Stanford University Computer Science Dept. Technical Report.

MacLennan, B. J., Fen - an axiomatic basis for program semantics, CACM 16, 8, August 1973, pp. 468-471.

Schoch, J., An overview of the language Smalltalk-72, Sigplan Notices 14, 9, September 1979, pp. 64-73.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12